Cairo University
Faculty of Engineering
Computer Eng. Dept.
CMPS103

Data Structures and Algorithms
LAB#3
Stacks, Queues, and Priority Queues

Objectives

After this lab, the student should be able to:

- Use class templates to implement stack ADT (array-based and linked list-based).
- Pass and return Stack (template) to/from functions
- Use class templates to implement queue ADT (array-based and linked list-based).
- Pass and return Queue (template) to/from functions
- Implement the copy constructor for the ADT implanted as linked list.
- Use class templates to implement priority queue
- Write code to use stacks/queues in some common applications.

Part I – Stack ADT

A stack is a LIFO (Last-In-First-Out) data structure that should support the following operations: push, pop, peek, and isEmpty.

The Stack ADT interface is as follows:

```
//  This is an updated version of code originally
//  created by Frank M. Carrano and Timothy M. Henry.
//  Copyright (c) 2017 Pearson Education, Hoboken, New Jersey.

template<typename T>
class StackADT
{
public:
   /** checks whether this stack is empty.*/
    virtual bool isEmpty() const = 0;

   /** Adds a new entry to the top of this stack.*/
   virtual bool push(const T& newEntry) = 0;

   /** Copies the top item of the stack to the passed parameter and removes it
from stack*/
   virtual bool pop(T& TopEntry) = 0;

   /** Copies the top item of the stack to the passed parameter without
removing it from stack*/
   virtual bool peek(T& TopEntry) const = 0 ;

   /** Destroys this stack and frees its assigned memory. */
   virtual ~StackADT() { }
}; // end StackADT
```

Stack Implementation:

StackADT can be implemented using an array or a linked list. Code example#1 gives an array- based implementation of the StackADT

🗐 Code Examples #1

Open "Stacks_Queues.sln" and run project "***1-Stacks***" and see how stack is implemented as class ***template*** using arrays:

1. Stack operations are implemented as class member functions
2. A checkpoint is provided inside the code. Look for it and follow its instructions

Stack as a function parameter

Stacks can be passed/returned to/from a function. They can be passed by value or by references like any other data types.

🗐 Code Examples #2

Run project "***2-Passing_Stack***" and see how stack can be passed and returned to/from a function.

1. Stack can be passed/returned by value, ref, const ref, etc.
2. Can we use pass by const ref instead of pass by value?
   Modify function PrintStack to be
   void PrintStack(const ArrayStack<T> &S).
3. What is the problem?

Part II – Queue ADT

A queue is a FIFO (**F**irst-**I**n-**F**irst-**O**ut) data structure that should support the following operations: enqueue, dequeue, peek, and isEmpty The queue ADT interface is as follows:

```
//  This is an updated version of code originally
//  created by Frank M. Carrano and Timothy M. Henry.
//  Copyright (c) 2017 Pearson Education, Hoboken, New Jersey.

template<class T>
class QueueADT
{
public:
   /** Sees whether this queue is empty.*/
   virtual bool isEmpty() const = 0;

   /** Adds a new entry to the back of this queue.*/
   virtual bool enqueue(const T& newEntry) = 0;

   /** Copies the front item of the queue to the passed parameter and removes
it from queue*/
   virtual bool dequeue(T& FrontEntry) = 0;

   /** Copies the front item of the queue to the passed parameter without
removing it from queue
   virtual bool peek(T& FrontEntry) const = 0;
   /** Destroys this queue and frees its memory. */
   virtual ~QueueADT() { }
}; // end QueueADT
```

Queue Implementation:

QueueADT can be implemented using an array or a linked list. Code example#3 gives a linked list-based implementation of the QueueADT

Code Examples #3

Run the project "***3-Queues***" and see how Queue is implemented as a class template using linked lists.

1. Queue is a class template and each Node in the queue is implemented by a class template as well.
2. Two pointers are needed: a front pointer (where dequeue operation takes place) and a back pointer (where enqueue operation takes place).

Queue as a function parameter and the copy constructor

Similar to stacks, queues can be passed/returned to/from a function. They can be passed by value or by references like any other data types.

Code Examples #4

Run project "***4-Passing_Queue***" and see how Queue can be passed to a function.

1. Why LinkedQueue is passed by reference to one function and by value to the other
2. Check how function QueueSum takes a template specialization of LinkedQueue class (i.e. ***LiknedQueue<int>***) not the generic template (***LinkedQueue<T>***).
3. A runtime error occurs when you run this code example. What is the problem and how to solve it?

Part III – Priority Queue

A priority queue is an abstract data-type similar to a regular queue data structure. Each element in a priority queue has an associated **priority**. In a priority queue, **elements with high priority are served** before elements with low priority. Typically, if two elements have the same priority, they are served in the same order in which they were enqueued (acts as normal queue in this case). In other implementations, the order of elements with the same priority is undefined.

Priority Queue Implementation:

While priority queues are often implemented using **heaps**, they are conceptually distinct from heaps. A priority queue is an abstract data structure like a list or a map; just as a list can be implemented with a linked list or with an array, a priority queue can be implemented with a **heap** or another method such as an **ordered list**.

Code Examples #5

Run project "***5-Priority Queue***" and notice:

1. How priority queue is implemented using sorted linked list. (The enqueue function inserts every new item in its ordered position relative to its priority)
2. In the main function: The comparison between priority queue and regular queue.

Exercises

Note:

In the following exercises, you are allowed to use only stacks, queues or priority queues as auxiliary memory if needed. You are NOT allowed to modify code of their classes. You should use them all as black boxes through their supported operations

1- Extend the code of project "***1-Stacks***" and declare a stack of names and test its member functions on the new data type.

2- Create a simple class Car that has Car number and engine number as members. Add setters and getters. Write a program that

1. Uses ArrayStack class template to instantiate an object of ArrayStack that would store **pointers** to Car.
2. Create 3 Cars and fill with arbitrary values and push them into the stack
3. Pop cars from the stack and print their info.

3- In code examples, Project "***2-Passing_Stack***"
Update function Reverse to be ***void ReverseMe(...stack to be reversed...)*** so that the function reverses the passed stack itself and does not return anything.

4- Write a function “collapse” that takes a stack of integers as a parameter and then collapses it by replacing each successive pair of integers by their sum.

For example:

- if a stack stores these values: bottom [**7, 2**, 8, 9, **4, 13**, 7, 1, **9, 10**] top. The stack should become: bottom [**9**, 17, **17**, 8, **19**] top.
- if a stack is bottom [1, 2, 3, **4, 5**] top. It becomes bottom[1, 5, **9**] top

5- The given code example shows how to implement the stack using arrays You are required to implement the stack using a linked list instead of an array.
Note: use class template ***Node*** given in Queue example.

6- A ***palindrome*** is a string that can read backward and forward with the same result.
e.g., the following is a palindrome: **MADAM**
Further, a more general palindrome is one that ignores spacing, punctuation, and capitalization, such as the following is also a palindrome: **Go dog and Madam, I’m Adam**
Write a function that checks if a given string is a general palindrome.

7- Write a function **print_reverse** that prints a linked list in reverse order. **(Do not use recursion).**

8- Write a function **copyStack** that copies the contents of one stack into another. The function takes two stacks, the source stack and the destination stack. The order of the stacks must be identical.

9- Write algorithm **catStack** that takes 2 stacks s1 and s2 and concatenates the contents of s2 on top stack s1. **Note:** s2 is unchanged. For example, if s1 = {1, 2, 3 top} and s2 = {4, 5 top} then s1 should be: {1, 2, 3, 4, 5 top} and s2 is unchanged.

10- Given a queue of **sorted** integers, write a C++ function that deletes duplicates from the queue **without changing the order of the remaining elements.** The original queue should be modified to contain only non-duplicates

11- The given code example shows how to implement the Queue using linked lists You are required to implement the Queue using arrays. What is the problem with array implementation? What is a circular queue? Implement it using array-based implementation

12- Write a function **SameOrder** that takes one stack and one queue and checks if the contents of both of them are in the same order. Same order means that both stacks and queues have the same number of items and the 1st item that **entered** the stack is the same as the 1st that **entered** the queue, and the 2nd is the same as the 2nd and so on.

13- One of the applications of a stack is to **backtrack**—that is, to retrace its steps. Write a program that:

1. Reads values from the user and stores them in a **queue** until zero is entered.
2. Scans the queue and does the following
   a. Each time a negative number is entered, the program must backtrack and print the five numbers that come before the negative number (starting from the one previous to the negative number) and then discard the negative number.
   b. If there are fewer than five items in the stack, print an error message and stop the program.
   c. The program repeats until the queue is empty or an error occurs

Test it with the following data:
Input Queue: **80** 1 2 3 4 5 **-1** **20** 30 40 50 **6 7 8 9 10** **-2** 11 12 -3 **33 22** **-5**

Output: 5 4 3 2 1 **10 9 8 7 6** 12 11 50 40 30 **22 33 20 80** Error: less than 5 items left

14- Write a function **catQueue** that concatenates two given queues together. The second queue should be put at the end of the first queue.

15- Write a function **queueToStack** that creates and returns a stack from a given queue. At the end of the algorithm, the queue should be unchanged; the front of the queue should be the top of the stack, and the rear of the queue should be the base of the stack

16- Given a queue of integers, write a function **deleteNegatives** that deletes all negative integers without changing the order of the remaining elements in the queue.

17- Write a function **reverseQueue** that reverses the contents of a queue.

18- Create class Car with each car has model, car_num, price. Write a program that reads data of N cars from the user and store them in a priority queue so that car with the lowest price is found at the front of the queue. Then the program should print all cars sorted ascendingly by car price.

19-

Queue-based App : Bank Queueing System

After solving the following problem on paper, write code to simulate the system operation.

System Description

Assume we have a bank with **one teller** working only at certain interval of time. Normally the arrival and service time for each customer should follow some pattern (usually random distribution) but for simplicity, here is the table of customer arrival and service times for each customer.

Name	Arrival Time	Service Time	Service Start Time	Service End Time	Waiting Time
A	3	12			
B	9	2			
C	16	8			
D	30	20			
E	34	10			
F	45	9			
G	47	5			

Table 1

Requirements:

1- Extend and fill the following table that shows the contents of the customers waiting queue with each enqueue or dequeue operation illustrating the time of the operation beside the queue.

Time	Operation	Waiting Queue content. Assume Front is on the left
0	Empty Queue	[ ]
3	Enqueue cust A	[A]
3	Dequeue A	[ ]

Table 2

2- Fill Table1 for each customer
3- Calculate average waiting time
4- Repeat the above problem but with two tellers in the bank and compare the new average waiting time to the old one

Notes for code implementation:

- Your system should be easily extensible to simulate different scenarios for N customers and M tellers (N and M could be loaded from a file or read from the user).
- It is preferable to generate arrival and service times randomly. For customer Names, you may replace with auto-generated unique IDs.
- Controlling N, M, and arrival and service time ranges would lead to different scenarios to simulate.
- You should print the system status every timestep. You should print info of tables 1&2 below in a suitable format
- Print the average waiting time and the average service time for every simulation scenario and compare the results.